Extending Hierarchical Delta Debugging with Hoisting

Dániel Vince, Renáta Hodován, Daniella Bársony and Ákos Kiss
Department of Software Engineering
University of Szeged
Szeged, Hungary
Email: {vinced,hodovan,bella,akiss}@inf.u-szeged.hu

Abstract—Minimizing failing test cases is an important pre-processing step on the path of debugging. If much of a test case that triggered a bug does not contribute to the actual failure, then the time required to fix the bug can increase considerably. However, test case reduction itself can be a time consuming task, especially if done manually. Therefore, automated minimization techniques have been proposed, the minimizing Delta Debugging (DDMIN) and the Hierarchical Delta Debugging (HDD) algorithms being the most well known. DDMIN does not need any information about the structure of the test case, thus it works for any kind of input. If the structure is known, however, it can be utilized to create smaller test cases faster. This is exemplified by HDD, which works on tree-structured inputs, pruning subtrees at each level of the tree with the help of DDMIN.

In this paper, we propose to extend HDD with a reduction method that does not prune subtrees, but replaces them with compatible subtrees further down the hierarchy, called hoisting. We have evaluated various combinations of pruning and hoisting on multiple test suites and found that hoisting can help to further reduce the size of test cases by as much as 80% compared to the baseline HDD. We have also compared our results to other state-of-the-art test case reduction algorithms and found that HDD extended with hoisting can produce smaller output in most of the cases.

Index Terms—test case minimization, hierarchical delta debugging, hoisting

I. INTRODUCTION

Our software can fail, so our software will fail. If we are lucky enough, we have a record of the events or inputs that triggered the failure. If we are even luckier, the failure is reproducible. In this case, a lucky engineer will get the task of fixing the problem.

Usually, the observed problem is only a symptom, and the root of it has to be found first in order to get the bug fixed. The record of the events or inputs that triggered the failure – i.e., the test case – can help here. However, this test case is often a mixture of relevant and irrelevant information. If much of the test case is irrelevant, i.e., it does not contribute to the failure, then the engineering time and effort required to fix the bug can increase considerably. Therefore, the minimization of failure-inducing test cases is an important first step on the path of debugging. However, it is of limited benefit if engineer-hours spent on bug fixing are simply converted to engineer-hours spent on manual test case reduction.

One of the most well-known techniques to automate reduction is the minimizing Delta Debugging algorithm (DDMIN)

by Zeller and Hildebrandt [1]–[3], working on all kinds of test cases without the need for any information about their structure. It has been realized, however, that if the structure of the test cases is known, that knowledge can be utilized to create smaller results faster. Misherghi and Su have introduced the Hierarchical Delta Debugging (HDD) algorithm [4], [5], built on DDMIN, that works on tree-structured inputs (e.g., on any input format that has a context-free grammar) and prunes unnecessary subtrees of the test case during reduction. These foundational works have inspired many follow-up research: some papers focused on improving their performance [6]–[9], while others aimed for smaller results [10].

In this paper, we also focus on the size aspect of the reduced test cases. We propose to extend the pruning-based reduction approach of HDD, where subtrees of the test case are removed, with a technique called hoisting, where subtrees are replaced with compatible subtrees further down the hierarchy. Therefore, we define the algorithmic framework of hoisting and describe its potential combinations with pruning. We have evaluated the introduced approaches and found that hoisting can help to further reduce the size of test cases by as much as 80% compared to the baseline HDD.

The rest of the paper is organized as follows: first, in Section II, we give a brief overview of DDMIN and HDD, to make this paper self-contained. Then, in Section III, we show some examples where pruning-based reduction can be improved upon, and describe and formalize the idea of hoisting. In Section IV, we evaluate the effects of hoisting with the help of a prototype implementation, and we present our experimental results. In Section V we discuss related work, and finally, in Section VI we summarize our work and conclude the paper.

II. BACKGROUND

The minimizing Delta Debugging (DDMIN) algorithm [1]–[3] is a systematic iterative approach for reducing a test case while keeping some interesting property invariant. The algorithm works on a set of atomic units representing parts of the test case. First, this set of units is split into two subsets of roughly equal size, and both subsets are investigated for whether they still have the interesting property of the original test case. If the property is kept in any of the subsets, reduction was successful and a new iteration starts with the found subset, otherwise the granularity is refined by doubling the splitting.

Let $test$ and $c_\mathbf{X}$ be given such that $test(\emptyset) = \checkmark \wedge test(c_\mathbf{X}) = \mathbf{X}$ hold.
The goal is to find $c'_\mathbf{X} = ddmin(c_\mathbf{X})$ such that $c'_\mathbf{X} \subseteq c_\mathbf{X}$, $test(c'_\mathbf{X}) = \mathbf{X}$, and $c'_\mathbf{X}$ is 1-minimal.
The *minimizing Delta Debugging algorithm* $ddmin(c)$ is

$$ddmin(c_\mathbf{X}) = ddmin_2(c_\mathbf{X}, 2) \text{ where}$$

$$ddmin_2(c'_\mathbf{X}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \dots, n\} \cdot test(\Delta_i) = \mathbf{X} \text{ (``reduce to subset'')} \\ ddmin_2(\nabla_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1, \dots, n\} \cdot test(\nabla_i) = \mathbf{X} \text{ (``reduce to complement'')} \\ ddmin_2(c'_\mathbf{X}, \min(|c'_\mathbf{X}|, 2n)) & \text{else if } n < |c'_\mathbf{X}| \text{ (``increase granularity'')} \\ c'_\mathbf{X} & \text{otherwise (``done'').} \end{cases}$$

where $\nabla_i = c'_\mathbf{X} - \Delta_i$, $c'_\mathbf{X} = \Delta_1 \cup \Delta_2 \cup \dots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_\mathbf{X}|/n$ holds.
The recursion invariant (and thus precondition) for $ddmin_2$ is $test(c'_\mathbf{X}) = \mathbf{X} \wedge n \leq |c'_\mathbf{X}|$.

Fig. 1. The Minimizing Delta Debugging Algorithm.

The subsets of the new partitioning are investigated again, one by one, as well as their complements. I.e., it is checked whether keeping or removing any of the subsets leads to an interesting smaller test case. Again, if any of the investigated test cases keeps the property in question, it will be used as the input to the next iteration, otherwise the splitting is increased. The iteration continues until the granularity reaches the unit level, when it is proven to have found a so-called 1-minimal result, a local minimum where the removal of any single unit from the test case causes the loss of the interesting property.

The algorithm has its roots in the isolation of failure-inducing code changes, which is visible in its terminology. For the algorithm, a test case is composed of elementary changes or deltas, denoted as δ_i, whence the algorithm got its name. A set of changes is also called a configuration, usually denoted by c. The outcome of a program run on a specific configuration is determined by a testing function, and it can be either FAIL (also written as $\mathbf{X}$) if the test case induced the original failure, PASS (also written as $\checkmark$) if the test succeeds, or UNRESOLVED (written as $?$) if the result is indeterminate. The set of all changes, i.e., the initial configuration that triggers a failing run is denoted by $c_\mathbf{X}$. Although the algorithm is often applied to the simplification of program inputs where the term "change" is not an intuitive fit to the units of a test case (e.g., to characters or lines of a text file) and the algorithm also has use cases where the interesting property of a test case is not a program failure. Most authors, including us, follow the original notation for historical reasons. For the sake of completeness, Figure 1 gives Zeller and Hildebrandt's latest formulation of the minimizing Delta Debugging algorithm [3].

If a test case that is to be reduced has some mandatory structure over its units, which is quite typical for inputs to a program, DDMIN may work suboptimally. The configuration partitioning during the iterations may be completely unaligned with the boundaries of the structural elements of the input, leading to incorrectly formatted, non-reproducing, and thus useless test cases. The goal of the Hierarchical Delta Debugging (HDD) algorithm [4] is to avoid such superfluous steps by not testing format-breaking configurations. To achieve that goal, it works on hierarchical tree-structured input representations (e.g., on parse trees, abstract syntax trees, or XML DOM

```
1  procedure HDD(input_tree)
2      level ← 0
3      nodes ← TAGNODES(input_tree, level)
4      while nodes ≠ ∅ do
5          minconfig ← DDMIN(nodes)
6          PRUNE(input_tree, level, minconfig)
7          level ← level + 1
8          nodes ← TAGNODES(input_tree, level)
9      end while
10 end procedure
```

Fig. 2. The Hierarchical Delta Debugging Algorithm.

trees) and applies the minimizing Delta Debugging algorithm to the levels of a tree, progressing downwards from the root to the leaves.

The pseudocode formulation of HDD as defined by Misherghi and Su [4] is shown in Figure 2. In the algorithm, the auxiliary routine TAGNODES collects the nodes at a given level of the tree, then DDMIN is invoked on those nodes, and finally PRUNE applies the result of Delta Debugging to the tree. I.e., for HDD, configurations are sets of tree nodes at a given level and removal of a node causes the removal of the whole subtree rooted at that node. In a later variant of HDD, "pruning" of a node has been reinterpreted as its replacement with the minimal applicable syntactically correct fragment to reduce the number of test attempts at incorrectly formatted configurations even further [5]. If HDD is iterated until a fixed-point is reached, denoted as HDD*, it gives a 1-tree-minimal result, i.e., if any single node is removed from the tree, the new test case will not be interesting anymore.

III. HOISTING

Although HDD performs better on structured inputs than DDMIN, there is still room for improvement. Several improvements have already been proposed, often by preprocessing the tree representation HDD is working on, e.g., by hiding some tokens from HDD and DDMIN to reduce the number of nodes that have to be considered, by collapsing (a.k.a. squeezing) multiple nodes into one for the same reason [7],

```
int main() {
  if (1) {
    printf("Hello world!\n");
  }
}
```

(a)
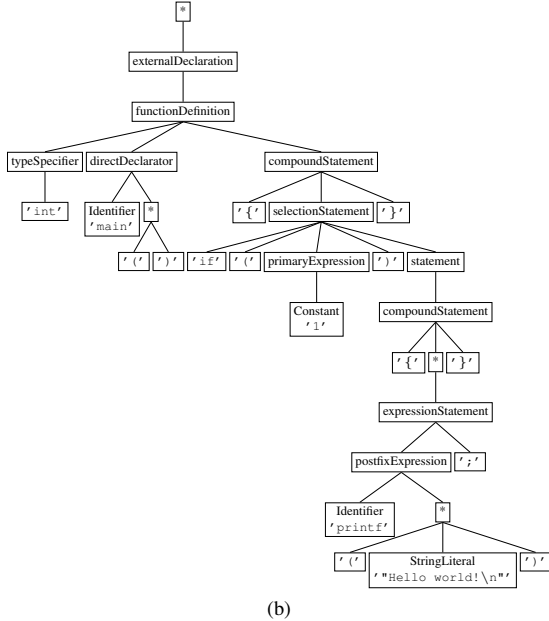
(b)

Fig. 3. An overly complicated "Hello World" program: (a) written in C and (b) its parse tree.

or by rotating recursive structures of the tree to reduce its height [8]. However, these transformations do not change the core structure of the tree, i.e., the test case generated (or, serialized) from the preprocessed tree will still be the same as the original input. Because of this, and because of the way HDD works, an HDD-reduced test case – although being 1-tree-minimal – may contain structural elements that a human expert would remove.

A. Examples

A simple example of this suboptimal structure-preserving behavior is shown in Figure 3. The C program in Figure 3(a) prints the classic "Hello world!" message, but it wraps the printing in an *if* statement where the predicate always evaluates to true. If we take this program as a test case and define the printing of the "Hello world!" message as interesting, then we can try and minimize it[1]. Figure 3(b) shows the parse tree of the program, generated by a parser using a context-free grammar of the C programming language and preprocessed for compactness (most notably, squeezing and recursion flattening have been applied). Unfortunately, HDD cannot reduce this test case any further (especially if replacement with minimal syntactically correct fragment is used when pruning subtrees) as removing any of the nodes would either yield a syntactically

[1]This is an example where the interesting property of the test case is *not* a program failure.
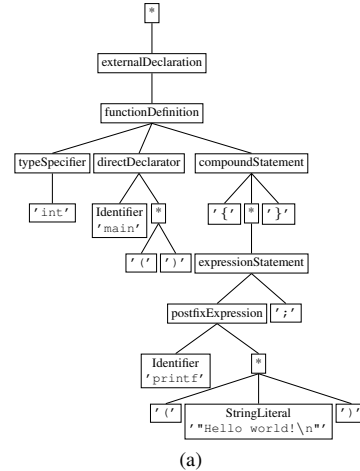
(a)

```
int main() {
  printf("Hello world!\n");
}
```

(b)

Fig. 4. The example program of Figure 3 minimized to keep printing the "Hello world!" message: (a) the parse tree with hoisting applied and (b) the C program serialized from the tree.

incorrect test case, or one that does not print the message, making it uninteresting[2].

Theoretically, both HDD and the underlying DDMIN algorithms could be modified to give n-tree-minimal results, but that would lead to exponential complexity, which is impractical. Thus, we propose another approach called *hoisting*.

We can observe that there are recurring structures in the parse tree, subtrees rooted at nodes with identical labels, denoting the derivation of the same non-terminal of the grammar. The assumption of hoisting is that one such subtree may be replaced with another without losing syntactic correctness, and that subtrees whose roots are in ancestor-descendant relationship may be good candidates for reduction. Of course, the testing function has to confirm (or reject) whether such a transformation keeps the resulting test case interesting.

In the tree of Figure 3(b), there is one pair of such subtrees, those rooted at nodes labeled as *compoundStatement*. Figure 4(a) shows a transformed tree where the descendant subtree is hoisted to replace all the structures that enclosed it. When this tree is serialized into the form of a C program (see Figure 4(b)), it becomes apparent that, in this case, this transformation was indeed useful and we got a smaller and still interesting test case.

When discussing the idea of hoisting with fellow researchers, the argument was often raised that such a transformation is only good for removing some minor syntactic elements from the result, like a dangling semicolon or a pair of superfluous braces, etc. The example of Figures 3 and 4 does not seem to contradict such arguments. However, Figure 5(a)

[2]DDMIN (either line or character-based) would not be able to remove the unnecessary *if* from around the printing either.

```
1  public class LocalizedPi {
2    private static String decSep(String locale) {
3      if (locale.equals("en")) {
4        return ".";
5      }
6      throw new Exception("Unsupported locale");
7    }
8    private static String formatParts(String intPart,
9        String fracPart, String decSep) {
10       return intPart.concat(decSep).concat(fracPart);
11   }
12   public static void main(String[] args) {
13     String pi = formatParts(
14       "3", "14", decSep(args[0]));
15     System.out.println(pi);
16   }
17 }
```

(a)

```
1  class LocalizedPi {
2    static String decSep(String a) {
3      throw new Exception("Unsupported locale");
4    }
5    static String formatParts(String intPart,
6        String fracPart, String decSep) {
7      return intPart.concat(decSep).concat(fracPart);
8    }
9    public static void main(String[] args) {
10     String a = formatParts("", "", decSep(args[0]));
11   }
12 }
```

(b)

```
1  class LocalizedPi {
2    static String decSep(String a) {
3      throw new Exception("Unsupported locale");
4    }
5    public static void main(String[] args) {
6      String a = decSep(args[0]);
7    }
8  }
```

(c)

Fig. 5. A program to print the rounded value of π in a locale-specific format: (a) written in Java, (b) minimized with HDD to keep the program throw an uncaught exception if an unsupported locale is specified on the command line, and (c) minimized with hoisting applied before HDD.

shows another example, a program written in Java, that prints the rounded value of π in a localized format, provided that the specified locale is supported, and throws an exception otherwise.

As presented, the program only supports the *en* locale. We shall take this program as a test case and the testing function shall check whether the program throws an exception when invoked with an unsupported locale (e.g., *hu*). If HDD is used to minimize this test case, it will be able to remove some parts of the program, but most of the original structure will remain in the output. (Because the exception that needs to be thrown is in the *decSep* method, which is called inside a call to the *formatParts* method, both methods are forced to be kept in the reduced test case.) The parse tree for this program would be too big to be presented as an example, so we only show the HDD-reduced Java program in Figure 5(b), which also displays precisely what HDD can and what it cannot prune away. However, if hoisting is used *before* HDD, it can pave the way for the latter reduction technique by hoisting the call to *decSep* to replace the enclosing call to *formatParts*, thus

allowing the complete removal of the definition of *formatParts*. (In this example, the method calls are the recurring structures that are in ancestor-descendant relation in the tree.) The result of the combined application of the two techniques is presented in Figure 5(c).

Note that *formatParts* could be constructed arbitrarily complex, making the theoretical potential of hoisting considerably higher than the removal of the anecdotical semicolons or curly braces. Also note, however, that hoisting cannot achieve this alone, as it "only" moves subtrees higher up the tree, but it has to cooperate with HDD.

B. Transformation-based Minimization

To formalize the ideas described and motivated above, we build on and extend the notations and terminology of delta debugging (as given in Section II) to introduce transformation-based minimization first.

In the context of delta debugging, a test case is always composed of a subset of the elements of the initial failing configuration. The testing function is also defined for the subsets of $c_{\bm{X}}$ only. However, it is useful (actually, necessary) if we can also determine the outcome of a program run on a set of elements, even if some of them are not part of the initial configuration (e.g., in the case of hoisting, when an element – a node – is replaced by another element – another node further down the hierarchy –, which is part of the tree, but is not a member of the initial set). Therefore, we extend the definitions of delta debugging [3] as follows.

Let D denote the set of all potential test case elements, and let $\delta \in D$ denote one element of that set, i.e., a test case element. A test case or configuration is denoted as $c \subseteq D$. A testing function $test : 2^D \to \{\bm{X}, \checkmark, ?\}$ shall determine for any test case whether it produces the failure in question. The initial failing configuration is denoted as $c_{\bm{X}} = \{\delta_1, \ldots, \delta_n\} \subseteq D$, and $test(c_{\bm{X}}) = \bm{X}$ holds.

As $c_{\bm{X}}$ is a subset of a potentially larger set D, we allow for *transformations* that can not only remove, but also *replace* elements in the configuration. We use the following definitions and notations for transformations:

A function $t : D \to D$ is a transformation of test case elements, and the identity transformation of test case elements is $id_D : D \to D; \delta \mapsto \delta$. We also define the application of a test case element transformation to test cases (or configurations) as $\bar{t} : 2^D \to 2^D; c \mapsto \{t(\delta) : \delta \in c\}$ (e.g., $\overline{id_D}(c_{\bm{X}}) = c_{\bm{X}}$). And a transformation that is derived from another transformation by changing the mapping of one test case element is defined as

$$t[\delta' \mapsto \delta''] : D \to D; \delta \mapsto \begin{cases} \delta'' & \text{if } \delta = \delta' \\ t(\delta) & \text{otherwise.} \end{cases}$$

In the examples of Section III-A, the transformations that could be applied were quite straightforward. There was only one *compoundStatement* and one method call that could potentially replace their parents. In a general case, however, a test case element may have multiple replacement candidates (or

Let D denote the set of all potential test case elements, and let $\delta \in D$ denote one element of that set.
Let $test$ and $c_{\boldsymbol{x}} = \{\delta_1, \ldots, \delta_n\} \subseteq D$, and $test(c_{\boldsymbol{x}}) = \boldsymbol{X}$ holds.
Let τ and $\|\cdot\|$ be given such that $\forall \delta \in D \cdot \forall \delta' \in \tau(\delta) \cdot \|\delta'\| < \|\delta\|$ holds.
The goal is to find $t_{\boldsymbol{x}} = tmin^{\tau}(c_{\boldsymbol{x}})$ such that $test(\bar{t}_{\boldsymbol{x}}(c_{\boldsymbol{x}})) = \boldsymbol{X}$ and $t_{\boldsymbol{x}}$ is 1-maximal.
The *transformation-based minimizing algorithm* $tmin^{\tau}(c)$ is

$$tmin^{\tau}(c_{\boldsymbol{x}}) = tmin_2^{\tau}(c_{\boldsymbol{x}}, id_D) \text{ where}$$

$$tmin_2^{\tau}(c_{\boldsymbol{x}}, t'_{\boldsymbol{x}}) = \begin{cases} tmin_2^{\tau}(c_{\boldsymbol{x}}, t'_{\boldsymbol{x}}[\delta \mapsto \delta']) & \text{if } \exists \delta \in c_{\boldsymbol{x}} \cdot \exists \delta' \in \tau(t'_{\boldsymbol{x}}(\delta)) \cdot test(\bar{t}'_{\boldsymbol{x}}[\delta \mapsto \delta'](c_{\boldsymbol{x}})) = \boldsymbol{X} \\ t'_{\boldsymbol{x}} & \text{otherwise.} \end{cases}$$

The recursion invariant (and thus precondition) for $tmin_2^{\tau}$ is $test(\bar{t'}_{\boldsymbol{x}}(c_{\boldsymbol{x}})) = \boldsymbol{X}$.

Fig. 6. The Transformation-based Minimizing Algorithm.

none at all). This is formalized using a function $\tau : D \to 2^D$ that maps test case elements to their transformed candidates.

Finally, as test cases are not necessarily the subsets of the initial failing configuration, minimality cannot be defined in terms of the subset relation anymore. Thus, we expect a $\|\cdot\|$ measure to exist on set D.

If all transformation candidates in τ are potentially reducing the size of a configuration according to the measure $\|\cdot\|$, i.e., $\forall \delta \in D \cdot \forall \delta' \in \tau(\delta) \cdot \|\delta'\| < \|\delta\|$ holds, then in order to minimize the test case we have to maximize the replacements applied to the elements of the initial configuration (even transitively) while ensuring that the so-transformed test case remains interesting. Just like it is true for DDMIN that searching for the global optimum is impractical, so is it also true for transformation-based minimization. Therefore, our actual goal is to find a local optimum, a *1-maximal* transformation $t_{\boldsymbol{x}}$ such that $\forall \delta \in c_{\boldsymbol{x}} \cdot \forall \delta' \in \tau(t_{\boldsymbol{x}}(\delta)) \cdot test(\bar{t}_{\boldsymbol{x}}[\delta \mapsto \delta'](c_{\boldsymbol{x}})) \neq \boldsymbol{X}$ holds.

Figure 6 wraps up this subsection and formalizes the transformation-based minimizing algorithm TMIN$^{\tau}$, worded in the likeness of DDMIN.

C. Hoisting and HDD

The transformation-based minimizing algorithm gives us a framework to formulate hoisting. The key to this is to define hoisting as a transformation of tree nodes. More precisely, to define those nodes in the tree representation of the input that can act as replacement candidates for their ancestors. The formula in Figure 7, $\chi(n)$, is one possible way to define these candidates, i.e., the hoistable descendants of a node n. $\chi(n)$ is given in terms of two auxiliary functions, of which CHILDREN(n) is trivial, giving the direct descendants of a node, whereas COMPATIBLE(n, n') leaves some space for interpretation. In an extreme case, any two nodes could be considered compatible, but that would rarely be useful. If the tree representation of the input is built using a context-free grammar, as motivated in Section III-A, then a natural interpretation is to regard identically-labeled nodes (i.e., subtrees of derivations of the same non-terminal of the grammar) as compatible.

The natural measure to use for tree nodes is based on the size of subtrees, i.e., the number assigned by the measure to

$$\chi(n) = \bigcup_{n' \in children(n)} \chi'(n, n')$$

$$\chi'(n, n') = \begin{cases} \{n'\} & \text{if } compatible(n, n') \\ \bigcup_{n'' \in children(n')} \chi'(n, n'') & \text{otherwise} \end{cases}$$

Fig. 7. $\chi(n)$, the potentially hoistable descendants of node n.

```
1   procedure HOIST(input_tree)
2       level ← 0
3       nodes ← TAGNODES(input_tree, level)
4       while nodes ≠ ∅ do
5           hoisting ← TMIN^χ(nodes)
6           TRANSFORM(input_tree, level, hoisting)
7           level ← level + 1
8           nodes ← TAGNODES(input_tree, level)
9       end while
10  end procedure
```

Fig. 8. The Hoisting Algorithm.

a node n equals the number of nodes in the subtree of n. It is obvious that all transformation candidates returned by $\chi(n)$ reduce the size of the configuration according to this measure, as expected by the definition of TMIN.

Now, with the help of TMIN$^{\chi}$, we can introduce a hierarchical algorithm, called HOIST, that works its way through the tree from the root to the leaves, and uses TMIN$^{\chi}$ to find the hoisting transformations at each level. Candidates found by TMIN$^{\chi}$ are prioritized by their distance to the ancestor, further nodes give higher priority. The pseudocode of the algorithm is presented in Figure 8. The structure of HOIST is similar to that of HDD: both contain a loop through the levels of the tree, and inside the loop, both perform a minimization step (TMIN$^{\chi}$ vs. DDMIN) and the application of its result to the tree (via the TRANSFORM and PRUNE auxiliary functions).

As discussed at the example of Figure 5, although HOIST can achieve reduction on its own, it is expected to work best if used in combination with HDD, e.g., by using HOIST as a preprocessing step. However, inspired by the similarities

```
 1   procedure HDDH(input_tree)
 2       level ← 0
 3       nodes ← TAGNODES(input_tree, level)
 4       while nodes ≠ ∅ do
 5           minconfig ← DDMIN(nodes)
 6           PRUNE(input_tree, level, minconfig)
 7           hoisting ← TMIN^X(minconfig)
 8           TRANSFORM(input_tree, level, hoisting)
 9           level ← level + 1
10           nodes ← TAGNODES(input_tree, level)
11       end while
12   end procedure
```

Fig. 9. The Hierarchical Delta Debugging & Hoisting Algorithm.

between the two algorithms, we can think of other ways of combination as well. E.g., the bodies of the loops of the two algorithms can be interlaced, performing both the DDMIN and TMIN^X-based minimizations at each level. One way to formulate this idea is shown in Figure 9, in the algorithm named HDDH.

IV. EXPERIMENTAL RESULTS

To evaluate the effects of hoisting, we have implemented the above introduced HOIST and HDDH algorithms and published them in the 21.3 release of the Picireny project[3]. Picireny is a hierarchical test case reduction framework written in Python that supports ANTLR v4[4] grammars, and already contains an implementation of the HDD algorithm. In this implementation of hoisting, we consider nodes labeled with the same non-terminal of the grammar as compatible, as discussed in Section III-C.

We have used the implemented algorithms in four different combinations during our experiments:

- HDD*: *hierarchical delta debugging without any hoisting, acted as the baseline.*
- HOIST*+HDD*: *hoisting was applied as a preprocessing step to HDD*.*
- HDDH*: *hoisting interlaced with HDD*.*
- HOIST*+HDDH*: *HOIST* and HDDH* algorithms are not mutually exclusive, thus we have used them in sequence.*

In all cases, the asterisk superscript denotes the fixed-point iteration of the marked algorithm.

As inputs, we have collected test cases from different sources. The first (small) set of test cases is composed of the examples from Section III-A (*helloworld.c* and *LocalizedPi.java*, with properties to keep as described above). The second set of test cases have already been used in the literature for benchmarking reduction: the Perses Test Suite[5] contains fuzzer-generated C sources that cause various internal

[3]https://github.com/renatahodovan/picireny
[4]https://github.com/antlr/antlr4
[5]https://github.com/perses-project/perses

compiler errors in the Clang and GCC compilers[6]. Finally, as a third set, we have composed a new suite of JavaScript sources, also generated with fuzzing, that cause failures in the JerryScript lightweight JavaScript engine (which will be referred to as the JerryScript Reduction Test Suite)[7]. In the case of the latter two test suites, the interesting property of the test cases to keep during reduction is the failure they induce.

For each test case, we have built its parse tree representation using the grammar available for the input format from the official ANTLR v4 grammars repository[8]. Before evaluating any of the reduction algorithm combinations, we have applied squeezing of linear tree components [7] and flattening of recursive structures to the trees [8]. The experiments were executed on a workstation equipped with an Intel Core i5-9400 CPU clocked at 2.9GHz and 16GB RAM. The machine was running Ubuntu 20.04 with Linux kernel 5.4.0.

Table I shows the sizes of the example test cases, both before and after reduction. In all cases, size is expressed as the number of non-whitespace characters to avoid bias from indentation or other formatting differences. For these example inputs, all reduction approaches that use hoisting (i.e., HOIST*+HDD*, HDDH*, and HOIST*+HDDH*) give a smaller output than the baseline HDD*. Actually, they all give exactly the same output for each input, reducing the size of *helloworld.c* by 16% and *LocalizedPi.java* by 48%, compared to HDD*. There is a difference in the performance of the algorithms though – measured in steps, i.e., how many times the testing function was invoked to determine the outcome of a test case –, as shown in Table II. For both examples, HOIST*+HDD* and HOIST*+HDDH* required fewer steps than HDD* to minimize the input, and HDDH* was also faster on *LocalizedPi.java*. However, HDDH* on *helloworld.c* executed more steps than the baseline. (In each row of these tables, as well as in the tables to follow, bold numbers highlight the best result(s).)

Tables III and IV show the results of the algorithms on the Perses Test Suite. Note that for the reduction of the test cases in this test suite, we have used two additional non-HDD-based state-of-the-art test case reduction tools to give a more comprehensive evaluation of hoisting. The two tools are Perses[5] (revision 34d4dc4, a Java and Kotlin-based implementation of the algorithm of Sun et al. [11]) and Pardis[9] (revision b656c6f, by Gharachorlu and Sumner [12]). For details on the algorithms and their implementations, the reader is referred to the corresponding papers and tool documentations.

As shown in Table III, all hoisting-based algorithm combinations produce a smaller output than the baseline in 17 of 19 cases. (For one input, *gcc-70127*, HDD* ran out of

[6]The Perses Test Suite comes with a docker environment provided by its authors. The environment is presumed to contain all compiler versions and tools required to reproduce the issue of each test case in the suite. However, that turned out not to be the case in practice. Thus, we have only used those test cases for evaluation that worked as expected at the time of writing this paper.
[7]https://github.com/vincedani/jrts
[8]https://github.com/antlr/grammars-v4
[9]https://github.com/golnazgh/PARDIS

TABLE I
EXAMPLES: INPUT AND OUTPUT SIZES (NUMBER OF NON-WHITESPACE CHARACTERS)

Test	Input	HDD*	HOIST*+HDD*		HDDH*		HOIST*+HDDH*	
helloworld.c	42	42	**35**	*(-16.67%)*	**35**	*(-16.67%)*	**35**	*(-16.67%)*
LocalizedPi.java	446	359	**186**	*(-48.19%)*	**186**	*(-48.19%)*	**186**	*(-48.19%)*

TABLE II
EXAMPLES: NUMBER OF TEST EXECUTIONS

Test	HDD*	HOIST*+HDD*		HDDH*		HOIST*+HDDH*	
helloworld.c	32	**26**	*(-18.75%)*	51	*(+59.38%)*	**26**	*(-18.75%)*
LocalizedPi.java	638	**302**	*(-52.66%)*	588	*(-7.84%)*	304	*(-52.35%)*

TABLE III
PERSES TEST SUITE: INPUT AND OUTPUT SIZES (NUMBER OF NON-WHITESPACE CHARACTERS)

Test	Input	HDD*	HOIST*+HDD*		HDDH*		HOIST*+HDDH*		Perses	Pardis
clang-22382	65,786	582	489	*(-15.98%)*	**475**	*(-18.38%)*	489	*(-15.98%)*	509	1,027
clang-22704	597,827	168	164	*(-2.38%)*	165	*(-1.79%)*	**161**	*(-4.17%)*	246	724
clang-23309	118,178	3,582	1,486	*(-58.51%)*	1,677	*(-53.18%)*	**1,416**	*(-60.47%)*	1,943	5,188
clang-23353	94,734	374	354	*(-5.35%)*	592	*(+58.29%)*	351	*(-6.15%)*	**331**	397
clang-25900	245,065	1,562	986	*(-36.88%)*	**885**	*(-43.34%)*	888	*(-43.15%)*	943	1,970
clang-26350	378,160	1,613	778	*(-51.77%)*	**585**	*(-63.73%)*	760	*(-52.88%)*	1,220	1,777
clang-26760	588,548	586	595	*(+1.54%)*	**297**	*(-49.32%)*	582	*(-0.68%)*	345	825
clang-27747	409,083	419	406	*(-3.10%)*	**377**	*(-10.02%)*	415	*(-0.95%)*	415	765
clang-31259	137,161	2,174	814	*(-62.56%)*	947	*(-56.44%)*	**796**	*(-63.39%)*	1,339	1,519
gcc-59903	166,754	1,726	1,432	*(-17.03%)*	620	*(-64.08%)*	1,298	*(-24.80%)*	**487**	3,301
gcc-60116	218,223	3,788	1,185	*(-68.72%)*	1,152	*(-69.59%)*	**941**	*(-75.16%)*	1,823	3,852
gcc-61383	110,643	1,701	1,041	*(-38.80%)*	**844**	*(-50.38%)*	874	*(-48.62%)*	1,908	2,525
gcc-61917	254,742	1,764	575	*(-67.40%)*	885	*(-49.83%)*	**570**	*(-67.69%)*	557	2,371
gcc-64990	439,587	2,844	561	*(-80.27%)*	1,282	*(-54.92%)*	**551**	*(-80.63%)*	843	2,787
gcc-65383	125,221	1,027	543	*(-47.13%)*	490	*(-52.29%)*	**441**	*(-57.06%)*	474	1,281
gcc-66186	139,087	2,614	978	*(-62.59%)*	**977**	*(-62.62%)*	**977**	*(-62.62%)*	1,124	4,144
gcc-66375	191,827	2,963	1,446	*(-51.20%)*	1,439	*(-51.43%)*	**1,430**	*(-51.74%)*	1,594	3,918
gcc-70127	400,556	—	992	—	**915**	—	947	—	998	1,986
gcc-71626	14,465	168	**167**	*(-0.60%)*	**167**	*(-0.60%)*	**167**	*(-0.60%)*	**167**	169

TABLE IV
PERSES TEST SUITE: NUMBER OF TEST EXECUTIONS

Test	HDD*	HOIST*+HDD*		HDDH*		HOIST*+HDDH*		Perses	Pardis
clang-22382	14,699	9,910	*(-32.58%)*	12,955	*(-11.86%)*	9,997	*(-31.99%)*	2,325	**2,331**
clang-22704	10,540	21,094	*(+100.13%)*	10,474	*(-0.63%)*	21,180	*(+100.95%)*	**1,890**	4,705
clang-23309	24,630	16,025	*(-34.94%)*	19,833	*(-19.48%)*	15,828	*(-35.74%)*	4,147	**3,960**
clang-23353	14,598	30,114	*(+106.29%)*	14,662	*(+0.44%)*	30,182	*(+106.75%)*	**2,288**	2,324
clang-25900	14,766	9,983	*(-32.39%)*	12,510	*(-15.28%)*	9,865	*(-33.19%)*	**2,115**	2,431
clang-26350	16,789	18,851	*(12.28%)*	14,831	*(-11.66%)*	19,847	*(+18.21%)*	**3,976**	8,835
clang-26760	12,957	11,808	*(-8.87%)*	11,884	*(-8.28%)*	11,835	*(-8.66%)*	1,933	**1,828**
clang-27747	7,174	13,899	*(+93.74%)*	6,601	*(-7.99%)*	13,911	*(+93.91%)*	1,559	**1,545**
clang-31259	19,239	8,791	*(-54.31%)*	15,914	*(-17.28%)*	8,992	*(-53.26%)*	2,230	**2,107**
gcc-59903	18,935	12,554	*(-33.70%)*	18,381	*(-2.93%)*	12,345	*(-34.80%)*	3,879	**3,761**
gcc-60116	23,844	12,740	*(-46.57%)*	17,153	*(-28.06%)*	12,041	*(-49.50%)*	**4,394**	4,975
gcc-61383	17,286	11,802	*(-31.73%)*	15,350	*(-11.20%)*	11,984	*(-30.67%)*	**3,326**	4,275
gcc-61917	17,455	8,432	*(-51.69%)*	13,769	*(-21.12%)*	8,525	*(-51.16%)*	**2,802**	3,566
gcc-64990	19,624	10,533	*(-46.33%)*	17,565	*(-10.49%)*	10,548	*(-46.25%)*	**2,690**	3,799
gcc-65383	16,239	6,524	*(-59.83%)*	11,801	*(-27.33%)*	6,334	*(-61.00%)*	**2,097**	2,843
gcc-66186	16,181	13,771	*(-14.89%)*	13,930	*(-13.91%)*	13,762	*(-14.95%)*	**2,333**	3,693
gcc-66375	21,251	16,046	*(-24.49%)*	18,393	*(-13.45%)*	16,131	*(-24.09%)*	**2,932**	3,013
gcc-70127	—	15,699	—	18,330	—	15,974	—	**2,517**	2,527
gcc-71626	4,216	6,520	*(+54.65%)*	4,205	*(-0.26%)*	6,522	*(+54.70%)*	567	**274**

TABLE V

JERRYSCRIPT REDUCTION TEST SUITE: INPUT AND OUTPUT SIZES (NUMBER OF NON-WHITESPACE CHARACTERS)

Test	Input	HDD*	HOIST*+HDD*		HDDH*		HOIST*+HDDH*		Perses
jerry-3299	1,208	92	**89**	*(-3.26%)*	**89**	*(-3.26%)*	**89**	*(-3.26%)*	152
jerry-3361	1,520	97	**95**	*(-2.06%)*	**95**	*(-2.06%)*	**95**	*(-2.06%)*	136
jerry-3376	4,647	70	**35**	*(-50.00%)*	37	*(-47.14%)*	**35**	*(-50.00%)*	202
jerry-3408	2,100	62	**54**	*(-12.90%)*	**54**	*(-12.90%)*	**54**	*(-12.90%)*	102
jerry-3431	648	28	**27**	*(-3.57%)*	**27**	*(-3.57%)*	**27**	*(-3.57%)*	62
jerry-3433	652	**18**	18	*(0.00%)*	18	*(0.00%)*	18	*(0.00%)*	67
jerry-3437	4,623	34	**18**	*(-47.06%)*	**18**	*(-47.06%)*	**18**	*(-47.06%)*	33
jerry-3479	3,998	94	**89**	*(-5.32%)*	**89**	*(-5.32%)*	**89**	*(-5.32%)*	413
jerry-3483	326	**38**	38	*(0.00%)*	38	*(0.00%)*	38	*(0.00%)*	50
jerry-3506	2,735	**52**	52	*(0.00%)*	52	*(0.00%)*	52	*(0.00%)*	135
jerry-3523	2,802	63	**48**	*(-23.81%)*	**48**	*(-23.81%)*	**48**	*(-23.81%)*	120
jerry-3534	1,409	96	**80**	*(-16.67%)*	**80**	*(-16.67%)*	**80**	*(-16.67%)*	115
jerry-3536	592	123	**120**	*(-2.44%)*	**120**	*(-2.44%)*	**120**	*(-2.44%)*	135

TABLE VI

JERRYSCRIPT REDUCTION TEST SUITE: NUMBER OF TEST EXECUTIONS

Test	HDD*	HOIST*+HDD*		HDDH*		HOIST*+HDDH*		Perses
jerry-3299	176	228	*(+29.55%)*	192	*(+9.09%)*	251	*(+42.61%)*	**169**
jerry-3361	**144**	254	*(+76.39%)*	154	*(+6.94%)*	266	*(+84.72%)*	199
jerry-3376	119	412	*(+246.22%)*	**109**	*(-8.40%)*	422	*(+254.62%)*	266
jerry-3408	167	278	*(+66.47%)*	178	*(+6.59%)*	289	*(+73.05%)*	**160**
jerry-3431	**55**	185	*(+236.36%)*	70	*(+27.27%)*	192	*(+249.09%)*	92
jerry-3433	**18**	58	*(+222.22%)*	23	*(+27.78%)*	62	*(+244.44%)*	27
jerry-3437	49	49	*(0.00%)*	**48**	*(-2.04%)*	56	*(+14.29%)*	78
jerry-3479	233	576	*(+147.21%)*	**230**	*(-1.29%)*	592	*(+154.08%)*	291
jerry-3483	69	95	*(+37.68%)*	71	*(+2.90%)*	97	*(+40.58%)*	**64**
jerry-3506	**115**	248	*(+115.65%)*	122	*(+6.09%)*	251	*(+118.26%)*	338
jerry-3523	111	416	*(+274.77%)*	**83**	*(-25.23%)*	421	*(+279.28%)*	295
jerry-3534	173	197	*(+13.87%)*	**149**	*(-13.87%)*	200	*(+15.61%)*	243
jerry-3536	150	226	*(+50.67%)*	182	*(+21.33%)*	251	*(+67.33%)*	**93**

memory, but the variants with hoisting correctly finished the minimization. To avoid a biased interpretation of data, we do not consider the correctly performing variants better than the baseline in this case.) The reduced test cases can be up to 80.27%, 70.07%, and 80.63% smaller (using HOIST*+HDD*, HDDH*, and HOIST*+HDDH*, respectively) than the result of HDD*. The average improvement is 37.15%, 38.54%, and 39.82%, respectively. When comparing them to each other, HDDH* gives the smallest result in 8 cases, HOIST*+HDDH* produces the smallest output in 9 cases, while there is also a tie, where HDDH* and HOIST*+HDDH* produce exactly the same output. Furthermore there is another tie when all three approaches find the same (local) minimum. When Pardis and Perses are also included in the comparison, the results of the hoisting-based minimizations are still good enough. In 15 of the 19 test cases, at least one of HDDH* and HOIST*+HDDH* gives smaller results than Perses, and in 11 cases both of them are better. (The detailed comparison to Pardis is omitted as Perses gave smaller results than Pardis in all cases of this experiment.)

Regarding performance, Table IV shows that hoisting can have a positive effect on the number of overall test case evaluations, but not necessarily. HDDH* performed the minimization

of 17 inputs faster than the baseline HDD*, but in those approaches where hoisting was a preprocessing step (namely, in HOIST*+HDD* and HOIST*+HDDH*), this improvement is only visible in 13 cases. Moreover, not even HDDH* could overcome Perses or Pardis speed-wise: these two tools were the fastest to produce output for all test cases of the Perses Test Suite.

The experimental results measured on the JerryScript Reduction Test Suite are shown in Tables V and VI. On these 13 inputs, all algorithms work quite similarly with respect to the output size: there are many cases where some or all approaches give identical results. Still, in 10 cases, all hoisting-based approaches give strictly smaller output than the baseline (by 50%, 47.14%, and 50% in the best case of HOIST*+HDD*, HDDH*, and HOIST*+HDDH*, respectively), while in none of the other cases do they give worse results. On this test suite, the average improvement of the approaches over HDD* is 12.85%, 12.63%, and 12.85%, respectively. Performance-wise, however, this test suite gives significantly different results than the Perses Test Suite. In the vast majority of the cases, the application of hoisting increased the number of testing steps performed during reduction. HOIST*+HDD*, HDDH*, and HOIST*+HDDH* were slower than HDD* in 12, 8, and 13 of the

13 cases, respectively.

We wanted to perform an experiment with Perses and Pardis on this test suite as well. Although none of these tools have official support for JavaScript inputs at the time of writing this paper, we have managed to extend Perses with the help of the same ANTLR v4 grammar files as used with Picireny. (As Perses outperformed Pardis on the previous test suite, we have not spent resources on extending Pardis, eventually.) The results show that all hoisting-based approaches give strictly smaller outputs than Perses, while producing minimized output faster in 4 of 13 cases.

Based on the experimental data and observations above, we can conclude the following:

- *On both artificial and real-world inputs, hoisting combined with hierarchical delta debugging gives generally smaller, or at least as small outputs as hierarchical delta debugging alone. Bigger outputs are rare. Minimized test cases can be as small as $\frac{1}{5}$ of the output given by traditional hierarchical delta debugging.*
- *In most of the cases, hierarchical delta debugging interlaced with hoisting (HDDH*) gives smaller results than non-HDD-based state-of-the-art reduction techniques.*
- *The effect of hoisting on the performance of test case reduction is inconclusive. All approaches that use hoisting have shown both improvements and deteriorations in terms of speed in several cases.*

V. RELATED WORK

One of the first and most well-known work on automated test case reduction is Delta Debugging by Zeller and Hildebrandt [1]–[3], minimizing inputs of arbitrary format. The price of its generality is a potentially lowered performance because of format-breaking incorrect test cases generated and evaluated during the reduction process. To avoid syntactically broken intermediate test cases, Miserghi and Su proposed to use information about format encoded in context-free grammars, i.e., to convert test cases into a tree representation [4] and apply delta debugging to the levels of the tree. This Hierarchical Delta Debugging approach helped to remove parts of the test case that aligned with syntactic unit boundaries. As a further improvement, Miserghi proposed the concept of syntactically correct replacement for nodes of the tree representation that cannot be completely removed from the test case without causing syntax errors [5].

The formalization of HDD does not detail how to build the tree representation, but its first implementation used traditional context-free grammars to parse the input. To improve on this, Hodován et al. suggested to use extended context-free grammars for building the tree [10], thus creating more balanced representations, which could lead to smaller results and improved performance. They have also described various tree transformations with the same goal [7], [8].

Tree-based test case reduction does not necessarily mean subtree removal. Bruno suggested to use hoisting as an alternative transformation in his framework called SIMP [13], which was specifically designed to reduce database-related inputs.

In every reduction step, SIMP tried to replace a node with a compatible descendant. In a follow-up work that introduced the tool FlexMin, Morton and Bruno extended SIMP with Delta Debugging [14]. The main algorithm was the hoisting, while DDMIN was applied only to repeated structures, like lists (column names) and data (string literals). Instead of manually classifying the nodes into two parts, our algorithm tries to hoist every node if it has at least one compatible descendant.

Sun et al. combined the above approaches into their Perses framework [11]. In their work, they utilized quantifiers and normalized the parse tree producing grammars by rewriting recursive rules to use quantified expressions. This transformed grammar form was referred to as Perses Normal Form (PNF). During the reduction, they applied a worklist algorithm, in which non-terminals with more tokens were prioritized over nodes with less token descendants. In every step, a node was popped from the worklist and reduced according to its type: quantified nodes were reduced with DDMIN while hoisting was applied on non-quantified, regular ones. They also mentioned that the number of compatible nodes for hoisting can be enormous. Instead of collecting all candidates and trying to hoist the descendant that has the longest distance from its ancestor, they limited the search space with two constrains: (1) the number of nodes between ancestor-descendants is limited (4 was used in their evaluation); (2) if a compatible descendant has been found, the searching was not continued further down in the hierarchy.

Built upon the ideas introduced in Perses, Gharachorlu and Sumner [12] extended it in a new framework, named Pardis, with an improved queue prioritization algorithm. This algorithm only considered nullable, i.e., completely removable tree nodes and assigned weights based on a node own token weight instead of its parents. The other key difference was that eventually they eliminated the hoisting step, since they found it too expensive from performance perspective.

Herfert et al. [15] also combined subtree removal and hoisting in their Generalized Tree Reduction (GTR) algorithm but instead of analyzing a grammar to decide about the applicability of a certain transformation they learned this information from an existing test corpus. The search-based program repair work of Gazzola et al. [16] also mentions modifications on the abstract syntax tree, however transformations are given as predefined templates and could be performed randomly without the ancestor-descendant relationship.

The above mentioned works targeted textual inputs, but test case reduction can be applied to other scenarios as well. Several authors have minimized faulty event sequences originating from various sources: Scott et al. [17] minimized event sequences of distributed systems, Clapp et al. [18] aimed at Android GUI event sequences with a variant of DDMIN. Moreover, Delta Debugging was even used for the minimization of SMT solver formulas (Brummayer et al. [19]).

An interesting analogy between test case reduction and program slicing was recognized by Binkley et al [20]–[25]. They have realized that the concepts of slicing (e.g., the program to be sliced or the slicing criterion) can be reformulated as

concepts of test case reduction (e.g., the test case or the interestingness property, respectively). Their approach, called observation-based slicing, avoids the complexities of building a dependency graph representation of a program and can work purely at syntactic level. Although their approach is not DDMIN-based, the algorithms show similarities with the ideas of DDMIN and HDD.

We can consider the algorithms as phases in our implementation that process the input via tree traversal. The pruning phase is identical to HDD, while HOIST phase tries to hoist the furthest compatible descendant from the tree hierarchy. In addition, other transformations can be implemented easily as new phases into the framework.

VI. SUMMARY

In this paper, we have been focusing on the automated reduction of tree-structured test cases. We have proposed an extension to the well-established pruning-based hierarchical delta debugging technique. The key idea of the extension, called hoisting, is to allow moving up subtrees in the structure of the test case, thus reducing its size. We have defined an algorithmic framework for generic transformation-based minimizations and used it to formalize hoisting. We have also described various ways of how hoisting can be combined with hierarchical delta debugging.

We have prototyped the introduced algorithms and evaluated several combinations of hoisting and pruning on multiple test suites. The results of our experiments support that hoisting can improve the output of hierarchical delta debugging by producing as much as 80% smaller minimized tests in the best case. Moreover, the prototype implementation gave smaller results than other, non-HDD-based state-of-the-art test case reduction tools in most of the experiments.

As for future work, we have plans to continue this topic of research in various ways. We wish to conduct further experiments to ensure that the results generalize to inputs that are larger or differently structured compared to those investigated in this paper. We plan to investigate the combination of hoisting with other HDD algorithm variants, like Coarse HDD [8] or HDDr [9]. We also aim at speeding up hoisting-extended HDD while not increasing the output size. Furthermore, we would like to explore additional minimizing transformations. Finally, we are also interested in the human understandability aspect of reduced test cases, considering both pruning- and transformation-based reduction methods.

ACKNOWLEDGMENT

REFERENCES

[1] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *Proc. 7th Eur. Softw. Eng. Conf. Held Jointly With 7th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (ESEC/FSE)*, ser. Lecture Notes in Comput. Sci., vol. 1687. Springer, 1999, pp. 253–267.

[2] R. Hildebrandt and A. Zeller, "Simplifying failure-inducing input," in *Proc. 2000 ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*. ACM, 2000, pp. 135–145.

[3] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, 2002.

[4] G. Misherghi and Z. Su, "HDD: Hierarchical delta debugging," in *Proc. 28th Int. Conf. Softw. Eng. (ICSE)*. ACM, 2006, pp. 142—-151.

[5] G. S. Misherghi, "Hierarchical delta debugging," Master's thesis, Univ. California, Davis, 2007.

[6] R. Hodován and Á. Kiss, "Practical improvements to the minimizing delta debugging algorithm," in *Proc. 11th Int. Joint Conf. Softw. Technol. (ICSOFT)*, vol. 1. SciTePress, 2016, pp. 241–248.

[7] R. Hodován, Á. Kiss, and T. Gyimóthy, "Tree preprocessing and test outcome caching for efficient hierarchical delta debugging," in *Proc. 12th IEEE/ACM Int. Workshop Automat. Softw. Test. (AST)*. IEEE, 2017, pp. 23–29.

[8] ——, "Coarse hierarchical delta debugging," in *Proc. 33rd IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*. IEEE, 2017, pp. 194–203.

[9] Á. Kiss, R. Hodován, and T. Gyimóthy, "HDDr: A recursive variant of the hierarchical delta debugging algorithm," in *Proc. 9th ACM SIGSOFT Int. Workshop Automating Test Case Design, Selection, Eval. (A-TEST)*. ACM, 2018, pp. 16–22.

[10] R. Hodován and Á. Kiss, "Modernizing hierarchical delta debugging," in *Proc. 7th Int. Workshop Automating Test Case Design, Selection, Eval. (A-TEST)*. ACM, 2016, pp. 31–37.

[11] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *Proc. 40th Int. Conf. Softw. Eng. (ICSE)*. ACM, 2018, pp. 361—-371.

[12] G. Gharachorlu and N. Sumner, "PARDIS: Priority aware test case reduction," in *Proc. 22nd Int. Conf. Fundam. Approaches Softw. Eng. (FASE)*, ser. Lecture Notes in Comput. Sci., vol. 11424. Springer, 2019, pp. 409–426.

[13] N. Bruno, "Minimizing database repros using language grammars," in *Proc. 13th Int. Conf. Extending Database Technol. (EDBT)*. ACM, 2010, pp. 382—-393.

[14] K. Morton and N. Bruno, "FlexMin: A flexible tool for automatic bug isolation in DBMS software," in *Proc. 4th Int. Workshop Test. Database Syst. (DBTest)*. ACM, 2011, pp. 1:1–1:6.

[15] S. Herfert, J. Patra, and M. Pradel, "Automatically reducing tree-structured test inputs," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*. IEEE, 2017, pp. 861—-871.

[16] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Trans. Softw. Eng.*, vol. 45, no. 1, pp. 34–67, 2019.

[17] C. Scott, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker, "Minimizing faulty executions of distributed systems," in *Proc. 13th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. USENIX Association, 2016, pp. 291–309.

[18] L. Clapp, O. Bastani, S. Anand, and A. Aiken, "Minimizing GUI event traces," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*. ACM, 2016, pp. 422–434.

[19] R. Brummayer and A. Biere, "Fuzzing and delta-debugging SMT solvers," in *Proc. 7th Int. Workshop Satisfiability Modulo Theories (SMT)*. ACM, 2009, pp. 1–5.

[20] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "ORBS: Language-independent program slicing," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*. ACM, 2014, pp. 109—-120.

[21] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "ORBS and the limits of static slicing," in *Proc. 15th IEEE Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*. IEEE, 2015, pp. 1–10.

[22] S. Yoo, D. Binkley, and R. Eastman, "Seeing is slicing: Observation based slicing of picture description languages," in *Proc. 14th IEEE Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*. IEEE, 2014, pp. 175–184.

[23] N. E. Gold, D. Binkley, M. Harman, S. Islam, J. Krinke, and S. Yoo, "Generalized observational slicing for tree-represented modelling languages," in *Proc. 11th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*. ACM, 2017, pp. 547—-558.

[24] D. Binkley, N. Gold, S. Islam, J. Krinke, and S. Yoo, "Tree-oriented vs. line-oriented observation-based slicing," in *Proc. 17th IEEE Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*. IEEE, 2017, pp. 21–30.

[25] ——, "A comparison of tree- and line-oriented observational slicing," *Empirical Softw. Eng.*, vol. 24, no. 5, pp. 3077–3113, 2019.